CS1020 Lecture Note #3: **Object Oriented Programming Part 2**

More OOP concepts

Lecture Note #3: **OOP Part 2**

- **Objectives:**
  - Introduce more predefined Java classes
  - Introduce inheritance via creating subclasses
  - Introduce generics, allow operations that are not tied to a specific data type

- **References:**
  - **Scanner** class:
    - Chapter 1, Section 1.7, pages 74 to 75
  - Wrapper classes:
    - Chapter 1, Section 1.1, pages 29 to 30
  - **Object** class:
    - Chapter 1, Section 1.5, pages 56 to 58
  - **Inheritance:**
    - Chapter 1, Section 1.4, pages 54 to 56
    - Chapter 9, Section 9.1, pages 480 to 490
  - **Generics:**
    - Chapter 9, Section 9.4, pages 499 to 507

Lecture Note #3: **OOP Part 2**

- **Programs used in this lecture**
  - TemperatureInteractive.java
  - ApproximatePI.java
  - TestMath.java
  - TestWrapper.java
  - TestInheritance.java
  - Ball.java and TestBall.java
  - BallV2.java and TestBallV2.java
  - TestPair.java
  - TestGenericPair.java
  - TestMoreGenericPair.java
- Download programs from module website:
  - http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html

Lecture Overview

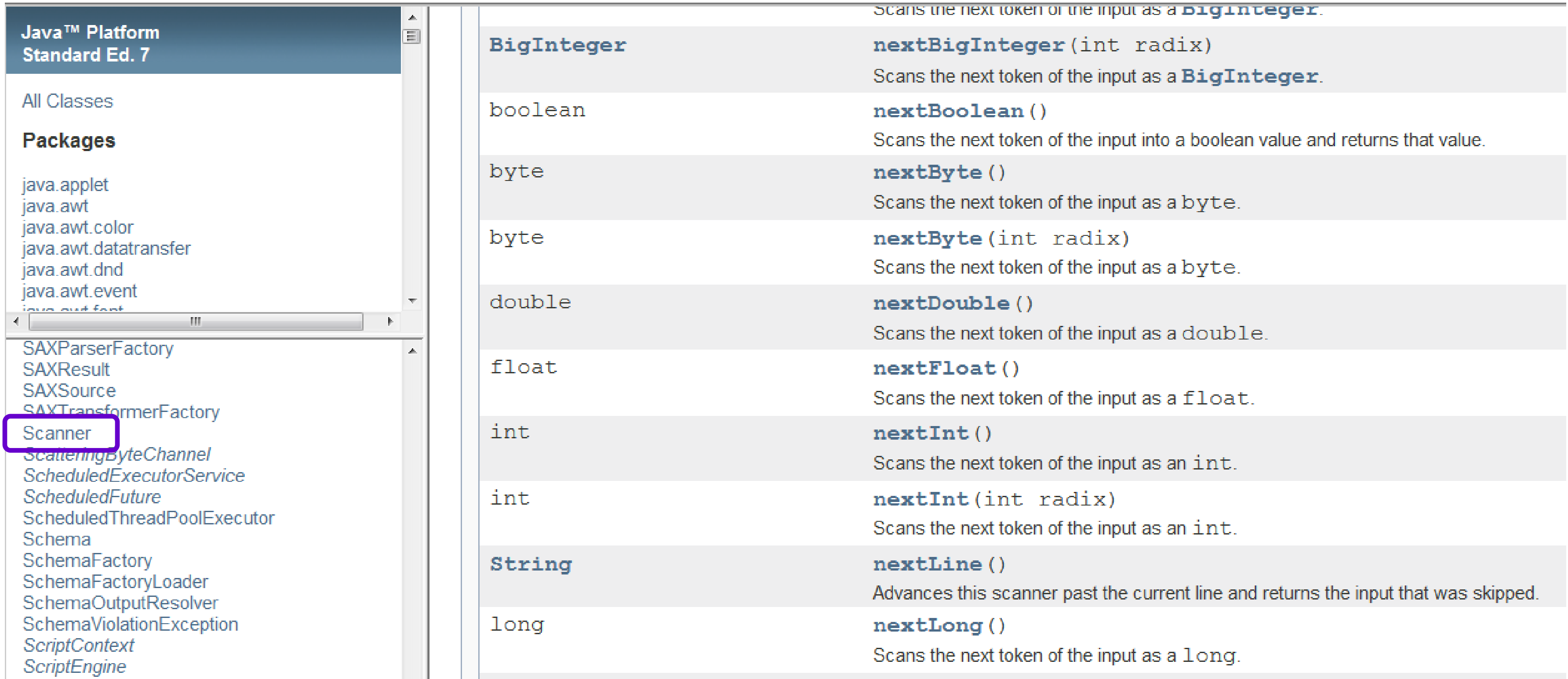
1. More Predefined Classes
   1.1 Scanner
   1.2 Math (introducing method overloading, static/class methods and class members)
   1.3 Wrapper Classes for Primitive Data Types
2. Inheritance
   - Method overriding
   - The Object class
3. Recapitulation
4. Generics

1. More Predefined Java Classes

- We introduced the `String` class last week.
- There are more predefined Java classes:
  - `Scanner` class
  - `Math` class
  - Wrapper classes
- Have you familiarised yourself with the Java API documentation?

1.1 The **Scanner** class (1/3)

- From the API documentation:



1.1 The **Scanner** class (2/3)

- In Lecture 1

TemperatureInteractive.java

```
import java.util.Scanner;

class TemperatureInteractive {

  public static void main(String[] args) {

     double fahrenheit, celcius;
     Scanner myScanner = new Scanner(System.in);

     System.out.print("Enter temperature in Fahrenheit: ");
     fahrenheit = myScanner.nextDouble();

     celcius = (5.0 / 9) * (fahrenheit - 32);
     System.out.println("Celcius: " + celcius);

  }

}
```

1.1 The **Scanner** class (3/3)

- In Lecture 1

ApproximatePI.java

```
import java.util.*; // using * in import statement

class ApproximatePI {
  public static void main(String[] args) {

      int i, nTerms, sign = 1, denom = 1;
      double PI = 0;

      Scanner myScanner = new Scanner(System.in);

      System.out.print("Enter number of terms: ");
      nTerms = myScanner.nextInt();

      for (i = 0; i < nTerms; i++) {
            PI += 4.0 / denom * sign;
            sign *= -1;
            denom += 2;
      }
      System.out.printf("PI = %.6f\n", PI);
  }
}
```

1.2 The **Math** class

- From the API documentation:

Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event

Marshaller
Marshaller.Listener
MaskFormatter
Matcher
MatchResult
Math
MathContext
MatteBorder
MBeanAttributeInfo
MBeanConstructorInfo
MBeanException
MBeanFeatureInfo
MBeanInfo
MBeanNotificationInfo
MBeanOperationInfo
MBeanParameterInfo
MBeanPermission
MBeanRegistration
MBeanRegistrationException
MBeanServer
MBeanServerBuilder
MBeanServerConnection
MBeanServerDelegate
MBeanServerDelegateMBean

Modifier and Type	Field and Description
`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a cir

Method Summary

Methods

Modifier and Type	Method and Description
`static double`	**`abs`**`(double a)` Returns the absolute value of a `double` value.
`static float`	**`abs`**`(float a)` Returns the absolute value of a `float` value.
`static int`	**`abs`**`(int a)` Returns the absolute value of an `int` value.
`static long`	**`abs`**`(long a)` Returns the absolute value of a `long` value.
`static double`	**`acos`**`(double a)` Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
`static double`	**`asin`**`(double a)` Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
`static double`	**`atan`**`(double a)` Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

1.2 The Math class

- Package: java.lang.Math (default)
- Some useful **`Math`** methods:
  - **`abs()`**
  - **`ceil()`**
  - **`floor()`**
  - **`max()`**
  - **`min()`**
  - **`pow()`**
  - **`random()`**
  - **`sqrt()`**
- Note the presence of many overloaded methods
  - **`abs(double a)`**, **`abs(float a)`**, **`abs(int a)`**, etc.

1.2 Method Overloading

- Overloading methods – 2 or more methods within the same class with the same name but different parameters
  - Very useful feature of Java
- Example: **abs()** method in Math class

```
public static int abs(int num)
```

Returns the absolute value of **num**.

```
public static double abs(double num)
```

Returns the absolute value of **num**.

- Hence, you may use **abs()** like this:

```
int num = Math.abs(-40);
double x = Math.abs(-3.7);
```

1.2 Method Overloading: Quiz (1/2)

- Given the following overloaded methods:

```
public static void f(int a, int b) {
   System.out.println(a + b);
}

public static void f(double a, double b) {
   System.out.println(a - b);
}
```

- What are the outputs of the following codes?

```
f(3, 6);

f(3.0, 6.0);

f(3, 6.0);
```

1.2 Method Overloading: Quiz (2/2)

- How about this?

```
public static void g(int a, double b) {
   System.out.println(a + b);
}
public static void g(double a, int b) {
   System.out.println(a - b);
}
```

- What is the output of the following code?

```
g(3, 6);
```

1.2 Static/Class methods

- Note that in the definition of every **`Math`** method, the keyword "**`static`**" appears.

`static double`	`sqrt(double a)` Returns the correctly rounded positive square root of a `double` value.	

- Such a method is called a static method (or class method).
- This means that no object (instance) of the **`Math`** class is required to use the method.
- Any **`Math`** method is called by preceding its name with the name of the class:
  - Example: **`Math.sqrt(area)`**

1.2 Class attributes

- The **`Math`** class also has two class attributes

`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

- A class attribute (or class member) is associated with the class, not the individual instances (objects). Every instance of a class shares a class attribute.
- How to use it?
  - Example: **`Math.PI`**

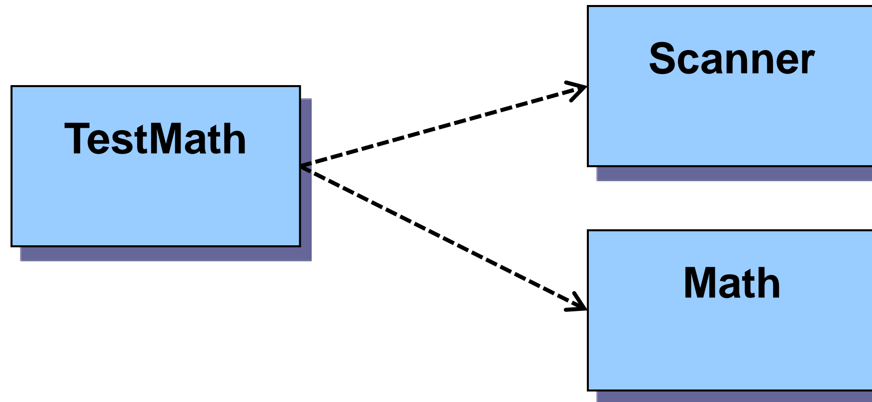
1.2 The **Math** class: Sample usage

TestMath.java

```
import java.util.*;

// To find the area of the largest circle inscribed
// inside a square, given the area of the square.

class TestMath {

  public static void main(String[] args) {

    double areaSquare, radius, areaCircle;

    Scanner myScanner = new Scanner(System.in);

    System.out.print("Enter area of a square: ");
    areaSquare = myScanner.nextDouble();

    radius = Math.sqrt(areaSquare) / 2;
    areaCircle = Math.PI * Math.pow(radius,2);

    System.out.printf("Area of circle = %.4f\n", areaCircle);
  }
}
```

1.2 Dependency Relationship

- The dependency relationship in **`TestMath.java`**



- **`TestMath`** class depends on both **`Scanner`** and **`Math`** classes.

1.3 Wrapper Classes: Motivation

- Other than the primitive data types, all other data in Java are in object form
  - Accessed through an object reference
  - Provide a number of methods and/or attributes

- The primitive data types are exceptions to the norm mainly due to efficiency considerations:
  - Object representation takes up more memory space
  - Object access is slower

1.3 Wrapper Classes: Motivation

- There are situations where we need an object representation of the primitive data types:
  - Java provides a number of wrapper classes for this purpose

Primitive Data Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`

1.3 Wrapper Classes Example: Integer

- From the API documentation:
  - Package: java.lang.Integer (default)

Java™ Platform Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color

InstantiationError
InstantiationException
Instrument
Instrumentation
InsufficientResourcesException
IntBuffer
Integer
IntegerSyntax
Interceptor
InterceptorOperations
InterfaceAddress
INTERNAL
InternalError
InternalFrameAdapter
InternalFrameEvent
InternalFrameFocusTraversalPolicy
InternalFrameListener
InternalFrameUI
InternationalFormatter
InterruptedByTimeoutException
InterruptedException
InterruptedIOException
InterruptedNamingException
InterruptibleChannel

	Returns the value of this `Integer` as a `byte`.
`static int`	**`compare`**`(int x, int y)` Compares two `int` values numerically.
`int`	**`compareTo`**`(Integer anotherInteger)` Compares two `Integer` objects numerically.
`static Integer`	**`decode`**`(String nm)` Decodes a `String` into an `Integer`.
`double`	**`doubleValue`**`()` Returns the value of this `Integer` as a `double`.
`boolean`	**`equals`**`(Object obj)` Compares this object to the specified object.
`float`	**`floatValue`**`()` Returns the value of this `Integer` as a `float`.
`static Integer`	**`getInteger`**`(String nm)` Determines the integer value of the system property with the specified name.
`static Integer`	**`getInteger`**`(String nm, int val)` Determines the integer value of the system property with the specified name.
`static Integer`	**`getInteger`**`(String nm, Integer val)` Returns the integer value of the system property with the specified name.
`int`	**`hashCode`**`()` Returns a hash code for this `Integer`.
`static int`	**`highestOneBit`**`(int i)` Returns an `int` value with at most a single one-bit, in the position of the highest-o
`int`	**`intValue`**`()` Returns the value of this `Integer` as an `int`.

1.3 Wrapper Classes: Sample usage

TestWrapper.java

```
class TestWrapper {

  public static void main(String[] args) {
    Integer intRefA, intRefB;
    int intPrimitive;

    intRefA = new Integer(4);
    intRefB = 4;

    if (intRefA == intRefB)
      System.out.println("Both refer to the same object");

    if (intRefA.equals(intRefB))
      System.out.println("Both contain the same value");

    intPrimitive = intRefA.intValue()
    intPrimitive = intRefB;
  }
}
```

`intRefA` and `intRefB` are references!

Object Instantiation

Alternative: Known as **auto boxing**

Conversion to primitive type

Alternative: Known as **auto unboxing**

2 Inheritance

Like father, like son

2.1 Inheritance : Motivation (1/3)

- Let's define a **saving account** class
- **Basic Information:**
  - account number, balance
  - **interest rate**
- **Basic Functionality:**
  - withdraw, deposit
  - **pay_interest**
- Compared with a basic account:
  - Differences are highlighted in bold font
  - Saving Account shares > 50% code with Bank Account
- Should we just cut and paste the code?

2.1 Inheritance : Motivation (2/3)

- Duplicating code is undesirable:
  - Hard to maintain
    - Need to correct all copies if error is found
    - Need to update all copies if modification is needed
    - etc.
- Since the classes are logically unrelated:
  - Code that work on one class cannot work on the other
  - Example:

```
public static void transfer(BankAcct fromAcct,
                            BankAcct toAcct, double amt);
```

will **not** work on **saving account** objects

- **Compilation error** due to incompatible data types

2.1 Inheritance : Motivation (3/3)

- Object oriented languages allow **inheritance**
  - Declare a new class based on an existing class
  - The new class **inherits** all of the attributes and methods from the other class

- **Terminology:**
  - If `class B` is derived from `class A`, then
    - `class B` is called a **child (subclass or derived class)** of `class A`
    - `class A` is called a **parent (superclass)** of `class B`

2.2 Saving Account : Implementation

```
class BankAcct {
  protected int _acctNum;
  protected double _balance;

  //Constructors and methods not shown

}
```

The "`protected`" keyword allows subclass to access the attributes directly

```
class SavingAcct extends BankAcct {

  protected double _rate;    //interest rate

  public void payInterest() {
      _balance += _balance * _rate;
  }

}
```

The "`extends`" keyword indicates inheritance

TestInheritance.java

2.2 Observations

- Inheritance greatly reduces the amount of redundant coding
  - No definition of **`account number`** and **`balance`**
  - No definition of **`withdraw()`** and **`deposit()`**
- Improve maintainability:
  - E.g. If the **`withdraw()`** method is modified in class **`BankAcct`**
    - no changes is needed in class **`SavingAcct`**
  - The code in class **`BankAcct`** remains untouched
    - Other programs using **`BankAcct`** are not affected

2.3 Constructors in subclass

- Unlike normal methods, constructors are **not inherited**
  - You need to define constructor(s) for the subclass

```
class SavingAcct extends BankAcct {

   protected double _rate;     //interest rate

   public SavingAcct(int aNum, double bal, double rate) {
        _acctNum = aNum;
        _balance = bal;
        _rate = rate;
   }

   //......payInterest() method not shown
}
```

TestInheritance.java

2.4 The "**super**" keyword

- The "**super**" keyword allows us to use the methods in the superclass directly
  - Including constructors
- If you make use of superclass's constructor:
  - Make sure it is the **first statement** in the method body

```
class SavingAcct extends BankAcct {

  protected double _rate;    //interest rate

  public SavingAcct(int aNum, double bal, double rate) {
      super(aNum, bal);
      
      _rate = rate;
  }
}
```

Use the constructor in BankAcct class

TestInheritance.java

2.5 Saving Accounts: Sample Usage

```
class BankAcct { ...... }   //not shown
class SavingAcct { ...... }   //not shown

class TestInheritance {

  public static void main(String[] args) {

     SavingAcct sa1 = new SavingAcct(2, 1000.0, 0.03);

     sa1.print();
     sa1.withdraw(50.0);

     sa1.payInterest();
     sa1.print();
  }
}
```

Inherited method from BankAcct

Method in SavingAcct class

TestInheritance.java

2.6 Method Overriding

- Sometimes we need to modify the inherited method:
  - To change / extend the functionality
  - This is known as **method overriding**
- In the `SavingAcct` example:
  - The `print()` method should be modified to include the interest rate in output
- To override an inherited method:
  - Simply recode the method in the subclass using the same method header
  - Method header refers to the name and parameters type of the method (also known as **method signature**)

2.6 Method Overriding: Example

```
class SavingAcct extends BankAcct {

  protected double _rate;    //interest rate

  public void payInterest() { …… }

  public void print() {
    System.out.println("Account Number: " + _acctNum);
    System.out.printf("Balance: $%.2f\n", _balance);
    System.out.printf("Interest: %.2f%%\n", _rate);
  }

}
```

- The first two lines of code is exactly the same as `BankAcct`'s `print()`:
  - Can we reuse `BankAcct`'s `print()` instead of recoding?

2.6 Use the "super" keyword again

- The "**super**" keyword can be used to invoke superclass's method as well
  - Useful when the inherited method is overridden

```
class SavingAcct extends BankAcct {

  protected double _rate;    //interest rate

  public void payInterest() { …… }

  public void print() {

    super.print();

    System.out.printf("Interest: %.2f%%\n", _rate);
  }
}
```

Use the `print()` in `BankAcct` class

2.7 Subclass Substitutability

- An added advantage for inheritance is that:
  - Whenever a super class object is expected, a sub class object **is acceptable as substitution!**
    - **Caution:** the **reverse is NOT true**
  - Hence, all existing functions that works with the super class objects will work on sub class objects with **no modification**!
- Analogy:
  - We can drive a car
  - Honda is a car (Honda is a subclass of car)
  - We can drive a Honda car

2.7 Subclass Substitution: Example

```
class BankAcct { ...... }  //not shown
class SavingAcct { ...... }  //not shown

class TestBankAcct {

  public static void transfer(BankAcct fromAcct,
              BankAcct toAcct, double amt) { ...... }

  public static void main( String[] args ) {
     BankAcct ba = new BankAcct(1, 234.56);
     SavingAcct sa = new SavingAcct(2, 1000.0, 0.03);

     transfer(ba, sa, 234.56);

     ba.print();
     sa.print();
  }
}
```

`transfer()` can work with `SavingAcct` object!

2.8 The "`Object`" class

- In Java, **all classes** are descendant of a predefined class called "**`Object`**"
  - **`Object`** class specifies some basic behaviors common to <u>all</u> objects
  - Any methods that works with **`Object`** reference will work on **object of any class**
  - Methods defined in the **`Object`** class are inherited in all classes
  - Two inherited **`Object`** methods are
    - **`toString()`** method
    - **`equals()`** method
  - However, these inherited methods usually <u>don’t work</u> (!) because they are not customised

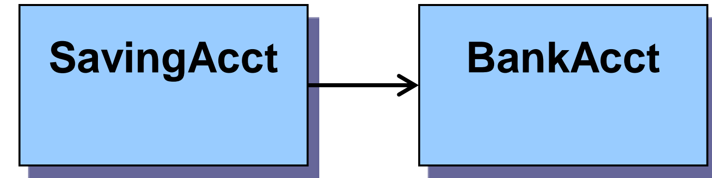
2.9 Pitfalls and Rules of thumb

- Beware:
  - Do not overuse inheritance
  - Do not overuse `protected`
    - Make sure it is something inherent for future sub class
- To determine whether it is correct to inherit:
  - Use the “**is-a**” rules of thumb
    - If “B is-a A” sounds right, then ***B is a subclass of A***
  - Frequently confused with the “**has-a**” rule
    - If “B has-a A” sounds right, then ***B should have an A attribute*** (hence B depends on A)

2.9 UML diagram: “is-a” and “has-a”

```
class BankAcct {
    ... ...
}

class SavingAcct extends BankAcct {
     ...  ...

}
```

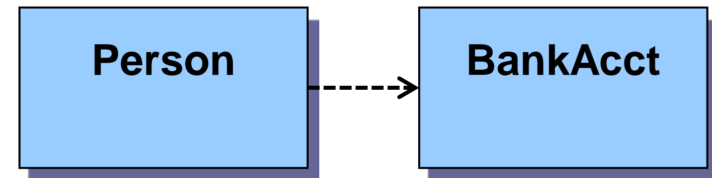
Inheritance: Saving Account IS-A Bank Account



Solid arrow

```
class BankAcct {
    ... ...
};

class Person {
     private BankAcct _myAcct;
};
```

Attribute: Person HAS-A Bank Account



Dotted arrow

3 Recapitulation

Let’s consolidate what we have learned so far

3.1 User-defined **`Ball`** class

- In this section we will create the **`Ball`** class to illustrate concepts covered:
  - Class and instance attributes
  - Overloaded constructors
  - Assessors and mutators
  - “this” keyword
- We will use **`BallV2`** class to illustrate
  - Overriding methods: **`toString()`** and **`equals()`**

3.1 Ball class (1/2)

Ball.java

```
// Version 1: basic
class Ball {
    /************** Data members **********************/
    // Assuming the inventory code for Ball is 12345
    private static int code = 12345;

    private String colour;
    private double radius;

    /************** Constructors **********************/
    public Ball() {
        setColour("yellow");  // default colour
        setRadius(10.0);      // default radius

        // the statements below work too
        // colour = new String("yellow");
        // radius = 10.0;
    }

    public Ball(String newColour, double newRadius) {
        setColour(newColour);
        setRadius(newRadius);

        // the statements below work too
        // colour = newColour;
        // radius = newRadius;
    }
```

Class attribute, shared by all objects of this class.

Instance attributes, owned by each instance (object).

Overloaded constructors

Could replace these 2 statements with:
`this("yellow", 10.0);`

3.1 **Ball** class (2/2)

Ball.java

```
    /**************** Accessors ***********************/
    public static int getCode() { return code; }

    public String getColour() { return colour; }

    public double getRadius() { return radius;}

    /**************** Mutators ***********************/
    // Why is "this" necessary here? How can the methods
    // be rewritten such that "this" becomes unnecessary?

    public static void setCode(int code) {
        Ball.code = code;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

}
```

3.1 TestBall program (1/2)

TestBall.java

```
import java.util.*;
class TestBall {
   public static void main(String[] args) {

       Scanner scanner = new Scanner(System.in);
       int inputCode;
       String inputColour;
       double inputRadius;

       // Create a default Ball object
       Ball myBall = new Ball();

       // What is myBall's code at this point?
       System.out.println("myBall's default code: " + myBall.getCode());

       // Read inputs from user
       System.out.print("Enter code: ");
       inputCode = scanner.nextInt();
       System.out.print("Enter colour: ");
       inputColour = scanner.next(); // What's difference between next()
                                     // and nextLine()?
       System.out.print("Enter radius: ");
       inputRadius = scanner.nextDouble();
```

3.1 TestBall program (2/2)

TestBall.java

```
        // Set the code, colour and radius of this Ball object
        // Note that we may call a static method on an instance:
        //    myBall.setCode(inputCode);
        // but this will be as good as the statement below
        Ball.setCode(inputCode);
        myBall.setColour(inputColour);
        myBall.setRadius(inputRadius);

        // Display the contents of the Ball object
        // Note also that we may call:
        //    myBall.getCode();
        // but again, it is as good as the statement below
        System.out.println("Code is " + Ball.getCode());
        System.out.println("Colour is " + myBall.getColour());
        System.out.println("Radius is " + myBall.getRadius());

        // What output do you get for the following statement?
        // (We will learn how to deal with it now.)
        System.out.println("Ball's contents are " + myBall);
    }
}
```

3.2 Overriding methods

- The **`Ball`** class inherited the **`toString()`** and **`equals()`** methods from **`Object`** class.
- The **`toString()`** is automatically invoked when an instance is printed:

Equivalent

```
System.out.println(myBall);
System.out.println(myBall.toString());
```

- Need to customise **`toString()`** and **`equals()`** to override the inherited ones
- We will create **`BallV2`** class and add the new codes.

3.2 Overriding methods: `toString()` & `equals()`

BallV2.java

```
// Version 2
class BallV2 {
   // omitted attributes, constructors, assessors, mutators

   /***************** Overriding methods *****************/
   // Overriding toString() method
   public String toString() {
      return "[" + getColour() + ", " + getRadius() + "]";
   }

   // Overriding equals() method
   public boolean equals(Object obj) {
      if (obj instanceof BallV2) {
         BallV2 ball = (BallV2) obj;
         return this.getColour().equals(ball.getColour()) &&
                this.getRadius() == ball.getRadius();
      }
      else
         return false;
   }
}
```

3.2 Overriding methods: TestBallV2 (1/2)

TestBallV2.java

```
import java.util.*;

class TestBallV2 {

   // This method reads ball's input data from user, creates
   // a ball object, and returns it to the caller.
   public static BallV2 readBall(Scanner sc) {

      System.out.print("Enter colour: ");
      String inputColour = sc.next();
      System.out.print("Enter radius: ");
      double inputRadius = sc.nextDouble();

      // Create a BallV2 object using the alternative constructor
      return new BallV2(inputColour, inputRadius);
   }

// Code continues to next slide
```

3.2 Overriding methods: TestBallV2 (2/2)

TestBallV2.java

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Read ball's input and create a ball object
        BallV2 myBall1 = readBall(scanner);
        System.out.println();
        // Read another ball's input and create a ball object
        BallV2 myBall2 = readBall(scanner);
        System.out.println();

        // Testing toString() method
        // How would output be like if there's no toString() in BallV2?
        System.out.println("1st ball: " + myBall1);
        System.out.println("2nd ball: " + myBall2);

        // Testing ==
        System.out.println("myBall1 == myBall2 is " +
                           (myBall1 == myBall2));

        // Testing equals() method
        System.out.println("myBall1.equals(myBall2) is " +
                            myBall1.equals(myBall2));
    }
}
```

3.3 User-defined **`ToyBall`** class

- Exercise: Write a **`ToyBall`** class which is a derived class (subclass) of **`Ball`**.
  - What are the new attributes you would like to add?
  - What are the new methods you would like to add?

4 Generics

Allowing operation on objects of various types

4.1 Generics: **Motivation**

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations
- In C, there is no easy way to exploit the similarity:
  - You need a separate implementation for each data type
- In Java, you can make use of **generic programming**:
  - A mechanism to specify solution <u>without</u> tying it down to a specific data type

4.2 Example: The Pair Class (non-generic)

- Let's define a class to:
  - Store a pair of integers, e.g. `(74, -123)`
  - **Many usages**: 2D coordinate, Range (min to max), Height and Weight, etc

```
class IntPair {

  private int _first, _second;

  public IntPair(int a, int b) {
    _first = a;
    _second = b;
  }

  public int getFirst() { return _first; }
  public int getSecond() { return _second; }
}
```

4.2 Usage: The Pair Class (non-generic)

```
import java.util.Scanner;

class IntPair { //definition not shown }

class TestPair {

  public static void main(String[] args) {

    IntPair range = new IntPair(-5, 20);
    Scanner sc = new Scanner(System.in);
    int input;

    do {
        System.out.printf("Enter a number in (%d to %d): ",
                range.getFirst(), range.getSecond());

         input = sc.nextInt();

    } while( input < range.getFirst() ||
                input > range.getSecond() );
  }
}
```

Sample usage of the `IntPair` class.

Purpose:
Repeatedly ask for user input until the input is in the desired range.

TestPair.java

4.2 Observation

- The `IntPair` class idea can be easily extended to other data types:
  - `double`, `String`, etc
- The resultant code would be almost the same!

```
class StringPair {

  private String _first, _second;

  public StringPair( String a, String b ) {
    _first = a;
    _second = b;
  }

  public String getFirst() { return _first; }
  public String getSecond() { return _second; }

}
```

Only differences are the data type declarations

4.3 Example: The Pair Class (Generic)

```
class Pair <T> {

  private T _first, _second;

  public Pair( T a, T b ) {
    _first = a;
    _second = b;
  }

  public T getFirst() { return _first; }
  public T getSecond() { return _second; }

}
```

The "<T>" is a formal generic type, user can **supply the desired data type later**

T is just like a variable, except that it stores a data type instead of a value

- Important restrictions:
  - The generic type can be substituted by **reference data type only**
    - **Primitive data types are NOT allowed**
  - Need to use Wrapper class for the primitive data type

4.3 Usage: The Pair Class (Generic)

```
class Pair <T> { //definition not shown }

class TestGenericPair {

  public static void main(String[] args) {

    Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
    Pair<String> twoStr = new Pair<String>("Turing", "Alan");

    //You can have pair of any reference data types!
    //..........
  }
}
```

TestGenericPair.java

- The formal generic type `<T>` is substituted with the actual data type supplied by the user:
  - The effect is similar to generating a new version of the **`Pair`** class, where **`T`** is substituted

4.4 Example: The Pair Class (V2.0)

- Let's modify the generic pair class such that:
  - Each pair can have two values of **different data types**

```
class Pair <S,T> {

  private S _first;
  private T _second;

  public Pair(S a, T b) {
    _first = a;
    _second = b;
  }

  public S getFirst() { return _first; }
  public T getSecond() { return _second; }

}
```

You can have multiple generic data types

Convention: Use capital single letter for the generic data type

4.4 Usage: The Pair Class (V2.0)

```
class Pair <S,T> { //definition not shown }

class TestMoreGenericPair {

  public static void main( String[] args ) {

    Pair<String,Integer> someone =
        new Pair<String,Integer>("James Gosling", 55);

    System.out.println("Name:  " + someone.getFirst());
    System.out.println("Age:  " + someone.getSecond());

  }
}
```

TestMoreGenericPair.java

- This `Pair` class is now very flexible!
  - Can be used in many ways

4.5 Generics: **Summary**

- Caution:
  - Generics is useful when the code remains unchanged other than differences in data type
  - When you declare a generic class/method, make sure:
    - The code is valid for all possible data types
- Additional Java Generics topics (not covered):
  - Generic methods
  - Bounded generic data types
  - Wildcard generic data types

4.6 Generics

- We will discuss the `Vector` class that involves the use of generics in the next lecture

Summary

Java Elements

Predefined classes
- Scanner class
- Math class
- Wrapper classes

Inheritance:
- Creating subclasses
- Overriding methods
- Object class

Generics:
- Defining a generic class
- Defining a generic class with multiple generic data type

End of file